NEKBONE: Thermal Hydraulics mini-application

Nekbone Release 2.1

May 15, 2013

Contents

1 Introduction to Nekbone **2**

2 Getting Started **3**

2.1 Setup 3

2.2 Running A First Example 3

3 Parameters in Nekbone **5**

3.1 SIZE File 5

3.1.1 Impact of Parameters 6

3.2 data.rea File 6

4 Details on Running and Editing Nekbone Examples **9**

4.1 Editing the Nekbone Test Example 9

4.2 Compiling Nekbone 10

4.3 Understanding the Output 11

4.3.1 Platform timer Results 11

4.3.2 Conjugate Gradient & Flop Counts 11

4.3.3 Bandwidth Test 12

5 The Nekbone Code: Default Setup **13**

5.1 The Default Setup 13

6 Nekbone & Nek5000 **15**

6.1 How Nekbone Represents Nek5000 15

6.2 MPI Communication within Nekbone 15

6.3 Optimization Opportunities 16

Chapter 1. Introduction to Nekbone

NEKBONE Release 2.1

Nekbone is captures the basic structure and user interface of the extensive Nek5000 software. Nek5000 is a high order, incompressible Navier-Stokes solver based on the spectral element method. It has a wide range of applications and intricate customizations available to users. Nekbone, on the other hand, solves a Helmholtz equation in a box, using the spectral element method. It is pared down to include only the necessary features to compile, run, and solve the applications found in the `test/` directory. Since almost all practical applications are in the three dimensional space, the solver is set to work with three dimensional geometries as default. Nekbone solves a standard Poisson equation using a conjugate gradient iteration with a simple preconditioner on a block or linear geometry (set within the test directory of the simulation). Nekbone exposes the principal computational kernel to reveal the essential elements of the algorithmic-architectural coupling that is pertinent to Nek5000.

More information about nekbone can be found on the CESAR website: **https://cesar.mcs.anl.gov/content/software/thermal_hydraulics** or by contacting one of the developers.


```
Paul Fischer    :  fischer@mcs.anl.gov
Katherine Heisey:  heisey@mcs.anl.gov
```


This document contains the quick start guide, an overview of the more detailed parameters available to the user, and a more detailed basis for the the connections between nekbone and Nek5000

Chapter 2. Getting Started

Nekbone requires the use of a F77 and C compiler. The currently tested and supported compilers are IBM, Intel, PGI Portland, GNU gfortran, although others may be used.

2.1 Setup

For the latest version of the nekbone code, please visit
https://cesar.mcs.anl.gov/content/software/thermal_hydraulics

After downloading the nekbone tarball, it can be unzipped and extracted in one step, if using the linux package, GNU tar commands:
`tar -zxvf nekbone-2.1.tgz`

This will create a nekbone-2.1directory populated with the source and test example directories.

Nekbone's test directory`(nekbone-2.1/test)` includes one example case for running Nekbone. `(nekbone-2.1/test/example1)`

Nekbone's source code is found in `nekbone-2.1/src/`.

An example must have a SIZE file and a data.rea file. (found in nekbone-2.1/test/example1/) Each test is compiled and linked with a makenek script. (Also in nekbone-2.1/test/example1/) This script performs a series of checks on the setup environment and compiler flags before compiling the source code using makefile.template to create the makefile.

For more details on this example and how to modify it, see section 4.1

2.2 Running A First Example

Change to this application's directory:
`cd nekbone-2.1/test/example1`

Check that the makenek script points to the correct source directory and edit it, if needed. The default is set to:
`SOURCE_ROOT='$HOME/nekbone-2.1/src'`

Check that the compiler is set as desired. The default compiler is set to a mpi wrapper for F77 and C. Change the `F77` and `CC` parameters in the makenek script found in `nekbone-2.1/test/example1/`

Compile the code using the makenek script to build and link:
`./makenek ex1` More details on the makenek script and how to modify it are found in Section 4.2.

A successful compilation of the code should result with this message printed to the screen:

```
################################################
#          Compilation successful!             #
################################################
```

and a nekbone executable in the `test/example1/` directory.

Running serial

To run the case in serial:
`./nekbone ex1`

Running in Parallel

To run the case in parallel the user can use the script provided, nekpmpi. The user must supply the name of the example and the number of processors to use, i.e:
`./nekpmpi ex1 4`
would run ex1 on 4 processors.

** NOTE: to run the application in parallel, one must be sure that the parameters set in the SIZE file accommodate the desired run parameters(specifically, *lp* and *lelt*). See section 3.1 for more details on these parameters.

To interpret the output, please see Section 4.3

Cleaning up

To clean up the source and test directory, removing the .o files, use:
`./makenek clean;`

Chapter 3. Parameters in Nekbone

To run an application, much like the standard Nek5000 examples, the user must run their experimental cases in a separate directory from where the code is stored. Each case ran with the nekbone code must have a SIZE file and a data.rea file in the running directory.

3.1 SIZE File

The SIZE file contains some basic parameters needed to create the mesh and control the parameter space. Below is a brief description of the parameters found in SIZE and how they can be changed to fit the user's needs. Most of the SIZE parameters are representative of the local processor counts as opposed to a global element representation.

ldim this is the dimension of the example. The code is written to work with three dimensions. Changing this parameter would produce unexpected results and is not recommended.

$lx1, ly1, lz1$ without being recompiled, this is the maximum polynomial degree set as $N = lx1 - 1$, where N is the polynomial degree. It can be any number, even or odd, that is greater than or equal to two. On some machine platforms, an advantage has been seen when using even numbers. However, on others there has been no evidence that either should be preferred. The parameters $lx1$, $ly1$, and $lz1$ should always be equal.

ldimt this parameter is used in the include parameter files and should be kept as is.

lp the maximum number of processors that can be used without recompiling the code. This parameter should be changed to reflect the MAXIMUM number of processors the user plans to run with.

lelt the maximum number of elements per processor that can be ran without recompiling the code. This should reflect the MAXIMUM number of elements per processor.

lelg the total number of elements in the run. This is set to be $(lp \times lelt)$ and should not be changed. The code is currently set to find the best

configuration across processors using this total number of elements in each dimension space.

lelx,*lely*,*lelz* this is the total number of elements in the x direction. Currently set to *lelg*. This parameter should not be changed.

common/dimn/ is the common block containing some of the most used variables in the code. Most are initialized in the beginning of the code and are equated to their counterparts named similarly. (i.e., $ndim = ldim$; $nx1 = lx1$; ect..) In general, these are the case specific parameters, not the bounding sizes.

3.1.1 Impact of Parameters

The parameters set in the SIZE file define the problem space to be evaluated. As stated above, *lelt* determines the number of total elements(per proc.) in the geometry where as $lx1, ly1, lz1$ define the polynomial order. As the figure 3.1 shows the polynomial order really enriches the geometry by increasing the total number of gridpoints.

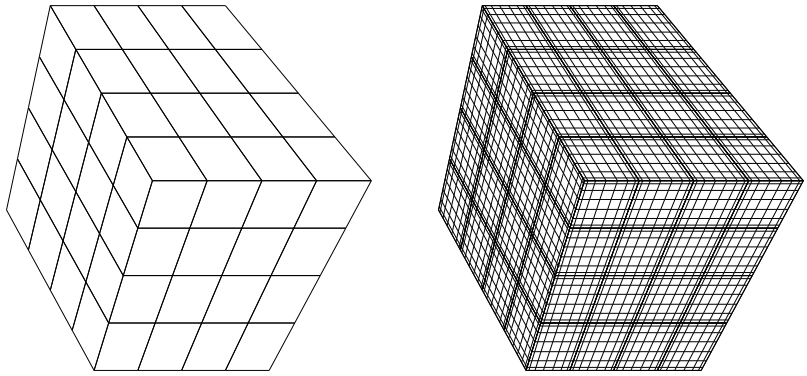

Figure 3.1: The role of $lx1$. The right geometry has no $lx1$ defined whereas the left has $lx1, ly1, lz1$ set to 7.

3.2 data.rea File

Along with the SIZE file, the data.rea file provides the user with a few parameters to be changed at runtime. This will allow users to alter certain variables without having to recompile the code.
EVERY EXAMPLE must have a data.rea file with these variables set:

ifbrick This is the logical switch used to determine a brick geometry or a linear block of elements. Setting `.true. = ifbrick` will allow the

code to determine the ideal 3-D configuration of nelt elements and np processors. Setting `.false. = ifbrick` will trigger the linear geometry. The linear geometry yields itself to an optimal communication pattern since each element only needs to communicate to 2 other elements on either side of it. (Excluding the ends, which would only have one neighbor) The brick configuration has a more realistic communication pattern where the interior elements need to communicate with 8 neighbors.

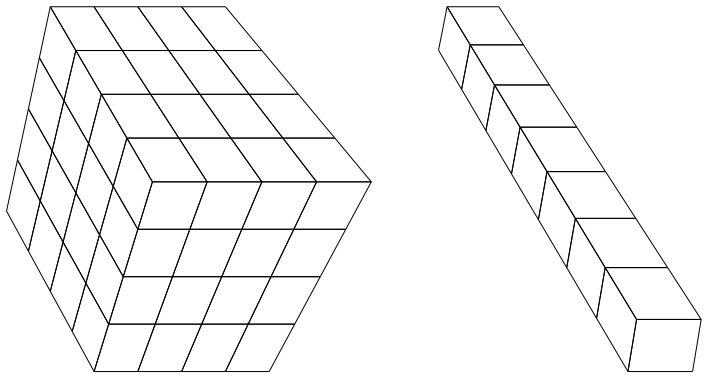

Figure 3.2: When `.ifbrick.` parameter is set to false a linear geometry is created (left) and when set to true, a 3-D brick of elements is created(right)

$iel0, ielN$ These two values are read in by nekbone and will control the range of elements to be evaluated, per processor. Nekbone will run a battery of tests starting with $iel0$ through $ielN$ elements per process. Thus, setting $iel0$ to 1 and $ielN$ to $lelt$ will loop through tests with 1 element per processor to $lelt$ elements per processor, as set in the SIZE file. If desiring a test that only runs a single Poisson solve on a specific element count, set $iel0 = ielN$. These values can be changed at runtime and do not require nekbone to be recompiled as long as $ielN <= lelt$.

$nx0, nxN$ In the current version, these parameters are not used. Future development will use these parameters to vary the polynomial order of a run. Currently, these are defaulted to be the $lx1$ value set in the SIZE file. These parameters will be read from the data.rea file and control the range of polynomial order. The polynomial order is set to be $nx1 - 1$ where $nx1$ is set according to $nx0$ and nxN. In coming revisions, setting $nx0 = 2$ and $nxN = lx1$ will run a series of tests from 2 through $lx1$ giving the full scope of polynomial ordering up to the maximum value set in SIZE. The default setup sets $nx0 = nxN = lx1$, therefore only running with a constant $nx1$

value equal to what is set in the SIZE file. Varying the polynomial order will change the computational complexity by increasing the number of grid points per element. Typical values span anywhere from 4 to 14, although much larger values have been explored.

Chapter 4. Details on Running and Editing Nekbone Examples

Nekbone's test directory (`nekbone/test`) includes one example case for running nekbone. (`nekbone/test/example1`)

4.1 Editing the Nekbone Test Example

Number of elements

This example will run a battery of problems in a single submission. Each problem increases in element count by the total number of processors being ran. Thus, the problem sizes can range from one element per process to $lelt$ elements per process, where $lelt$ is set in the SIZE file. This can be changed to range from any beginning and ending number of elements per processor by changing the parameters $iel0$ and $ielN$ in the data.rea file. As described in 3.2, these parameters control the range of tests to be ran as $iel0$ though $ielN$ as long as $iel0 > 0$ and $ielN <= lelt$. The default sets $iel0$ to 1 and $ielN$ to $lelt$, thus running a total of $lelt$ tests.

Varying Polynomial Order

The default setup of nekbone runs with a set polynomial order equal to $lx1$ as set in the SIZE file (see Section 3.1). In future development, Nekbone will be able to be configured to run a range of increasing polynomial orders by setting the parameters $nx0 >= 2$ and $nxN <= lx1$. See 3.2 for more details.

Naming Examples

In the current set up, the name of the example is not important and is not used. In future revisions, this might become integral to the code. However, as a basic set up, we have used the SIZE and data.rea parameters to specify the exact specifications. No mesh data or input data is read in besides there two files.

Geometry

Using the logical variable, $ifbrick$, found in data.rea, the user can control whether the geometry is set to be a brick or just a single line of elements. $ifbrick$ in this example is set to false, resulting a the optimal communication pattern that a linear geometry lends itself to.

In this example, the total degrees of freedom are
$dof = np \times nx1^3 \times nelt \leq lp \times lx1^3 \times lelt$.

number of CG iterations

The conjugate gradient solver is set to run for a maximum number

of iterations, niter. niter is set in `src/drive.f` and is can be increased as the degrees of freedom increase in the example. A lower niter value may result in non-converging results simply due not being allowed to iterate in the solver long enough for the degrees of freedom to be resolved.

4.2 Compiling Nekbone

Nekbone is compiled by running the provided script, `makenek`. Makenek allows the user to set the compiler, any compiler flags, optimization flags, and other preprocession flags.

SOURCE_ROOT One of the important variables that is defined in the script is the source directory path, `SOURCE_ROOT=`. This should be set to the path to the source code. Since the tests are all ran from their own directory, this path can be locally defined as
`../../src`
or more globably as the path from the user's HOME/ directory. As default, the path is set to
`$HOME/nekbone-2.1/src`
which assumes that the tarball was downloaded and unzipped in the HOME/ directory.

F77 and CC F77 and CC are the compilers to be used. Nekbone has been tested with GNU's gfortran, PGI Portland, INTEL and a few others. Both serial and parallel version have been used. The standard, mpif77 and mpicc are default in the test directory.

PPLIST PPLIST sets pre-defined pre-processor symbols that are used within nekbone. Currently, setting this variable to BG will enable some optimizations specific to Blue Gene P platforms.

IFMPI Uncommenting this variable sets IFMPI to false. As the name implies, this would turn off MPI communication within nekbone and enable a serial run. This should be toggled to false when using a serial compiler or when wanting to run without MPI enabled.

G The G variable is for any compiler flags the user wants to include. A common setting is compiling with debugging turned on by setting G = "-g". For PGI Portland serial compilers, adding -Ktrap=fp will cause the test to exit when encountering any NaN values.

Optimization Flags General optimization flags can be specified by setting the OPT_FLAGS_STD variable as desired. This will set the optimization level for a majority of the source files. If this is not specified, the code is compiled with $-O2$ and with $-O0$ when in debugging

mode.

OPT_FLAGS_MAG is used to set the highest level of optimization, which is used on some of the of the more intricate files. If this variable is undefined, these files with be compiled with $-O3$ and $-O0$ when in debugging mode.

Once the variables are defined as desired, running makenek in the test example directory:
`./makenek name_of_test`
will compile and link the code to be ran. See section 2 for more details on running the example provided in `nekbone-2.1/test/example`.

4.3 Understanding the Output

4.3.1 Platform timer Results

When the platform_timer(ivrb) is turned on, the result of all platform tests will be at the beginning of the logfile. This includes all_reduce times, varying times of different matrix- matrix product routines, and ping pong tests done on the platform ran.

4.3.2 Conjugate Gradient & Flop Counts

Nekbone writes to stdout the results of each conjugate gradient sequence on increasing problem size.

At the beginning of each sequence, nekbone prints:
`cg:  iter rnorm`
where iter should be 0, since the test is just beginning.

After *niter* iterations (set within `src/drive.f` of the nekbone source code), a summary of the convergence is printed to stdout.

`cg:iter rnorm alpha beta pap`

If the conjugate gradient iteration converged, *iter* is less than *niter* and *rnorm* should be close to the tolerance set.

After the conjugate gradient sequence is completed, the total flop count is printed to the screen.

`mflops flop_a flop_cg time1 flops nelt np nx1`

More detail is given in Section 5.1. Since the default implementation of nekbone is set to run increasing elements per processor, *np* remains constant and *nelt* should increase from 1 to *lelt* (set in SIZE). This is the number of elements per process. If $iel0$ and $ielN$ in data.rea (see 3.2) are not equal, $nx1$ should reflect that as well.

4.3.3 Bandwidth Test

If the bandwidth bisection test is turned on, nekbone will print the results of the gather-scatter routines using the crystal router exchanges done on an increasing number of points per process.

```
np npts npoints etime "bandwidth"
```

Where
np is the total number of processors
npts is the points per process exchanged
npoints is the total number of points in this test ($np \times npts$)
etime is the average time it took to exchange these points across processors.

This will test the rate of message transfers with increasing sized messages of the total number of processes.

Chapter 5. The Nekbone Code: Default Setup

Nekbone solves the Helmholtz equation in a box using the spectral element method. It partitions the computational domain into high-order quadrilateral elements. Based on the number of elements, number of processors, and the parameters of a test run, Nekbone creates a decomposition that is either a 1-dimensional array of 3D elements, or a 3-dimensional box of 3D elements. It evaluates a Poisson equation on every time step iteration and provides an estimate of realizable flops, as well as inter node latency and bandwidth measures.

5.1 The Default Setup

Boundary Conditions & Preconditioning

In order to ensure Dirichlet boundary conditions, a mask is applied in each conjugate gradient iteration. For simplicity this mask zeros out the first point on the first processor. This maintains a solvable code while not complicating the it with an extravagant masking mechanism. Similarly, the preconditioning step is the result of the identity matrix applied to the vector.

Platform Timing Tests

The default application of this code is set to run a battery of tests useful in profiling the platform and basic communication structure of the code. The first set of tests are called in driver.f by:

```
call platform_timer(iverbose)
```

The variable, iverbose, controls how much information is sent to standard output and can be flagged with a 0, for not verbose, or a 1 for verbose. These tests include ping-pong tests and all-reduce tests to give relevant information about the platform being ran on. Currently, this call is commented out to allow for short test runs.

Poisson Evaluation

After the platform timing tests, the Poisson equation evaluations begin. An iterative conjugate gradient solve is performed on an increasing number of elements per process from $iel0$ to $ielN$, set in the data.rea file. (See 3.2) The principle kernel of the code is the $\underline{w} = A * \underline{p}$ routine with has many opportunities for optimization. Essentially, the bulk of this work is done through matrix-vector

products. These $\underline{w} = A * \underline{p}$ evaluations are done on the local elements on each processor. To update across all processors, a nearest neighbor communication must be executed. The conjugate gradient evaluates iterate for a determined number of iterations, *niter*, set in driver.f.

Counting FLOPS Inside the conjugate solver, the flops are counted and timed for further analysis as the problem size grows. This counter is output to the logfile for each problem size as:

```
mflops flops_a flops_cg time 'flops' nelt np
```

where,for MPI processes running on rank 0,
mflops is the total_number of flops divide by time_spent in solver
flops_a is the operations spent in the Ax=b routines
flops_cg is the operation count spend in the conjugate gradient
time is the total time spend in the solver
nelt is the number of elements
np is the number of processors
nx1 is the value of nx1, polynomial order

Calculating Bandwidth Finally, the bandwidth of processors *np* can be tested with a call to:

```
call xfer(np,cr_h)
```

Here an array of increasing size is exchanged and timed across processors, averaged over 50 exchanges. This gives an idea of bisection bandwidth capacity of a range of data sizes. It is essentially testing the rate of message transfers with increasing sized messages, over the total number of processors. The default setting has this call commented out to speed up the overall time spent in any nekbone case.

Chapter 6. Nekbone & Nek5000

6.1 How Nekbone Represents Nek5000

As described above, nekbone is a conjugate gradient solver with a simple precondition implemented. Nek5000's temperature solve is a conjugate gradient iteration with multi-level point-Jacobi preconditioner. Any Nek5000 application that spends a majority of time in the temperature solver will very closely resemble a nekbone test ran on a large, brick element count. We have found that Nek5000 runs at parallel efficiency at ∼6,000-10,000 points per core. This means that the total degrees of freedom ($lelt \times lx1^3$) of a nekbone test should also follow this rule of thumb and one should expect to see similar results as nekbone scales to large processor counts. Also, both Nek5000 and nekbone's memory requirements scale as $lelt \times (lx1^3)$.

A Nek5000 Case with natural convection at high Rayleigh number ($Ra > 10^10$) will spend around 82% of the CPU time in the Helmholtz solve. Of this, 19% is spent in the precondition, which is not yet in nekbone. This leave 63% of the run time spent in calculations that are represented by the kernels found within nekbone. Since a principal challenge of exascale is to boost single-node performance, nekbone focuses on the main kernel in question.

6.2 MPI Communication within Nekbone

The communication kernel used in the standard Nek5000 software is the exact kernel used in this more basic code. nekbone communication is nearest-neighbor communication which is the majority of what is found in the Nek5000 application. Written primarily in C and C preprocessor, the communication routines are found in `nekbone-2.1/src/jl/`. The mini application accesses these routines to set up and exchange information across processors. The code is a parallel code, utilizing the MPI standard. Most MPI routines are employed through a wrapper found in comm_mpi.f.

6.3 Optimization Opportunities

Nekbone provides multiple levels of optimization. Since the bulk of the nekbone code focuses on the matrix-vector operations, this is a section of the code that could be highly optimized. Already, these routines have been optimized on most platforms common in the current computing resources.

The gradient kernel include 3 matrix-vector calls on the same data and the gradient-transpose kernel includes 3 matrix-matrix calls on different data to produce one output.